32-bit Floating Point Real & Complex 16-Tap FIR Filter Implemented Using Streaming SIMD Extensions

Version 2.1

01/99

Order Number: 243643-002

Table of Contents

Revision History

Revision	Revision History	Date
2.1	FCS revsion.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *Signals and Systems*, Alan V. Oppenheim and Alan S. Willsky (Copyright 1983, Prentice-Hall)

1. Introduction

The Streaming SIMD Extensions for the Intel®Architecture (IA) instruction set provide floating point single-instruction, multiple-data (SIMD) instructions. These instructions accelerate applications that rely heavily on operations that work on floating-point data (such as 3D graphics, real-time physics, and spatial audio). This application note discusses the implementation of both a real and complex 16-tap finite duration impulse response (FIR) filter using Streaming SIMD Extensions, and includes examples of code that exploit the SIMD instruction set.

This application note also provides a performance comparison between the various implementations presented for Streaming SIMD Extensions and other implementations available for Intel® processors. A 16-tap FIR filter implementation was chosen to provide a comparison of FIR benchmarks on other general purpose processors.

2. A 16-Tap FIR Filter

FIR, or moving average, filters are commonly used in Digital Signal Processing (DSP), and are typically found in image processing and audio algorithms. The filter performs a weighted average of neighboring points. The "weights" are referred to as the coefficients or taps of the filter. An M-tap FIR filter performs a weighted average of the previous M input points according to the following equation:

$$y(n) = \sum_{k=0}^{M} b_k \, x(n-k)$$

In the equation given above, x(n) represents an input vector of length N, y(n) represents an output vector of length N, b_k is the kth (of M) coefficients, and x(n) is zero for n less than zero.

An FIR filter can be used to implement a smoothing function or a high- or low-pass filter, depending on the choice of values for the taps.

2.1 Background for FIR Filters

The following is taken from the book *Signals and Systems* by Alan V. Oppenheim and Alan S. Willsky (Copyright 1983, Prentice-Hall). Please refer to this book, or another signal processing text, for a more complete discussion of filtering in general and the FIR filter in particular.

> "In a variety of digital signal processing applications it is of interest to change the relative amplitudes of the frequency components in a signal or perhaps eliminate some frequency components entirely, a process referred to as filtering. For linear time-invariant systems, the spectrum of the output is that of the input multiplied by the frequency response of the system. Consequently, the filtering can be conveniently accomplished through the use of such systems with an appropriately chosen frequency response.
>
> Low pass filtering can be thought of as a smoothing operation. For discrete-time sequences, a common smoothing operation is the moving average, where the smoothed values y(n) for any n_0 is an average of values of x(n) in the vicinity of n_0. One example of a weighted moving average filter is the finite-duration impulse response (FIR) filter."

2.2 Implementing the 16-Tap FIR Filter

This section presents two implementations of a 16-tap FIR filter: one implementation uses real numbers, the other uses complex numbers. Real FIR filters are commonly used general purpose filters in image processing, audio, and communications algorithms. Depending upon how a signal is processed, Complex FIR filters can be found in communications algorithms for echo cancellation and equalization. The basic FIR algorithm is the same for either case, but there are important differences in the implementation stemming from the complications of complex arithmetic and the additional storage requirements of complex numbers.

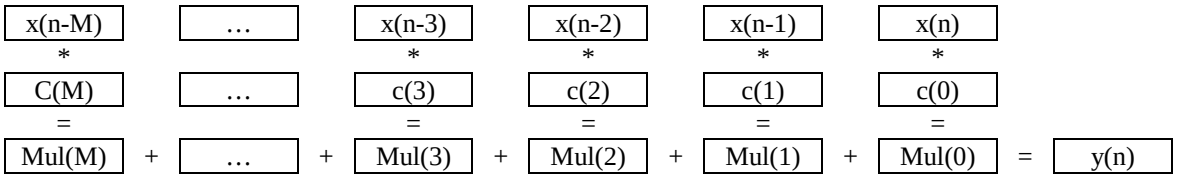
2.2.1 A 16-Tap FIR Filter for Real Numbers

This section discusses the design considerations that are appropriate for implementing the FIR filter for real numbers using the Streaming SIMD Extensions.

2.2.1.1 Design Decisions

Before an algorithm is implemented using Streaming SIMD Extensions, it must be examined for parallelism. The algorithm may be redesigned to take advantage of any inherent parallelism and can then be implemented using the SIMD instructions provided by Streaming SIMD Extensions.

To see the parallel, or SIMD, nature of the convolution at the heart of an FIR filter, first examine the following diagram:



Each y(n) is the sum of the multiplication of individual elements between the input and coefficients vectors (a dot product). It is immediately obvious that:

- Each of the multiplications is independent, and therefore can be done in parallel,
- Multiplication and addition are both associative and therefore, can be performed in small chunks and combined to produce a whole (*i.e.*, the dot product `a*b + c*d + e*f` can be accomplished by three multiplications performed in parallel and then two sets of two additions involving two values, `g = a*b + c*d & h = g + e*f`, or one addition involving three values).

Note: Due to rounding errors in 32-bit floating point arithmetic, rearranging mathematical operations may produce differing results in the least bits of significance, even for operations which are associative.

Often an algorithm may have multiple levels of parallelism which potentially can be exploited by SIMD techniques. In the case of the 16-tap FIR filter, the nature of the data used by the algorithm determines the best method for exploiting the filter's parallelism.

The Streaming SIMD Extensions operate on four 32-bit floating point numbers at a time. By working backward from the desired end result; that is, a register holding four 32-bit floating point numbers equal to four output values which can be written to the output array, one can see how the parallelism in the FIR algorithm is revealed.

	Bits 127 - 96	Bits 95 - 64	Bits 63-32	Bits 31-0
xmm7 =	out[n+3]	out[n+2]	out[n+1]	out[n]

Figure 1: Representation of a Pentium® III register

For each of the output values shown in Figure 1, a summation (or accumulation) of multiplications between the array of input values and the array of taps (or coefficients) must be calculated.

2.2.1.2 Parallelizing the Multiplications

Taking into account the opportunities for parallelism in the filter algorithm and the structure of the input data, the following argument can be made: if the array of coefficients was stored in reversed order, multiple multiplications (up to all 16 for a 16-tap filter) could be performed simultaneously, limited only by the number of data elements which can be operated upon at the same time (in the case of Streaming SIMD Extensions, this is four 32-bit floats).

Address:	low								high
Taps:	c(0)	c(1)	…	c(M-2)	C(M-1)	c(M)			
Reversed Taps:	c(M)	c(M-1)	…	c(2)	c(1)	c(0)			
	*	*		*	*	*			
Input:	x(0)	x(1)	…	x(M-2)	x(M-1)	x(M)	…	x(n-1)	x(n)

From the preceding example:

$$y(M) = x(M) * c(0) + x(M-1) * c(1) + \ldots + x(0) * c(M)$$

The multiplications are performed in parallel, followed by a summation of the results. To calculate the next output value, y(M+1), the taps array could be "slid" one element with respect to the input array, new products calculated and summed together, and so on.

If there are no restrictions on memory accesses, this "sliding" method of parallelizing the multiplications would be sufficient. However, Streaming SIMD Extensions place a 128-bit, or 16-byte, boundary restriction on loads, stores, and operations involving memory operands consisting of four 32-bit floats. The "sliding" method still can be taken advantage of using Streaming SIMD Extensions by storing four offset copies of the taps array as shown below:

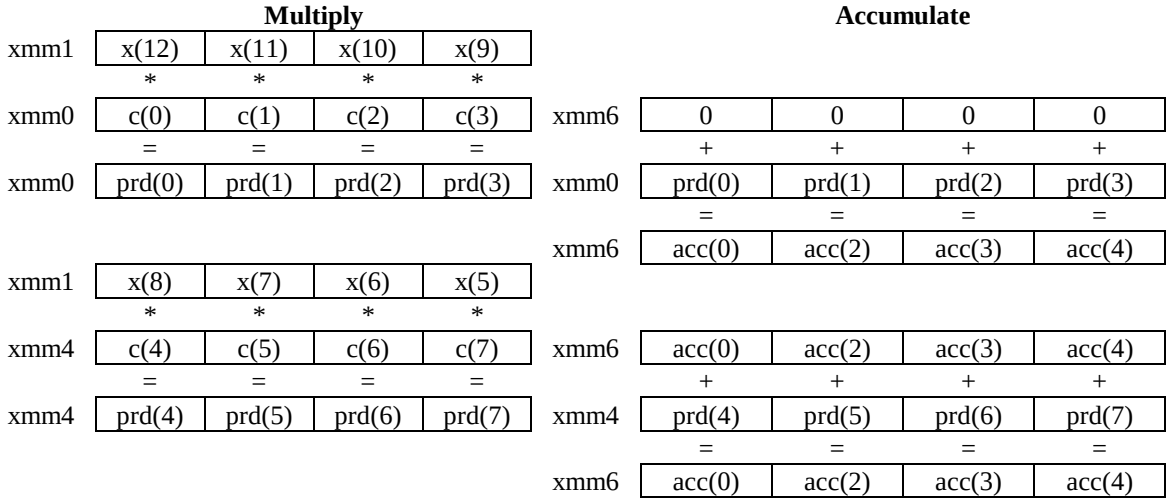
16 bytes				16 bytes				16 bytes				16 bytes				16 bytes			
c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0	0	0	0	0
0	c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0	0	0	0
0	0	c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0	0	0
0	0	0	c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0	0

Each group of four coefficients, representing four 32-bit floating point numbers, can be loaded into an xmm register. All four arrays of coefficients are allocated contiguously, so that as long as the entire block of memory is aligned to 16 bytes, each set of four coefficients is aligned to 16 bytes. Zero values are used to "mask" undesired products from the parallel multiplication. The "sliding" of the taps array with respect to the input array is achieved by working "down" the block of arrays, using different (horizontal) groups of the coefficient arrays to calculate out(n), out(n+1), out(n+2), and out(n+3).

Note: The configuration of the four tap arrays is entirely dependent on the number of taps (16 in this case). Given more or fewer taps, the result is a slightly different variation of the arrays (e.g. the upper right set of four coefficients will not be all zeroes for certain numbers of taps).

2.2.1.3 Parallelizing the Additions

An accumulator can be used to "collect" the products resulting from each of the multiplications for a particular output value. The accumulator actually collects four separate sums of products which, when summed together "across the register," yields the final output value. This is demonstrated below in calculating a portion of y(12):



Notice that the two multiplications are independent and, therefore, can be performed in parallel. After additional multiplications, the value of y(12) is obtained by summing the acc(x) values stored in register `xmm6`. This is basically a non-parallel operation performed by rearranging the values to enable addition, but the following paragraph discusses how the particulars of this implementation allow the summation to be parallelized.

It is not immediately obvious how additions across a register can be parallelized. In fact, summing across only one register is an inherently non-parallel operation. However, this can be overcome by collecting multiple accumulators (in this instance, four) and interleaving their addition using the Streaming SIMD Extensions unpack-high (`unpckhps`) and unpack-low (`unpcklps`) instructions. For each iteration of the FIR filter, products for four output values, out(n) through out(n+3), are accumulated in separate registers and then the accumulators are summed as follows.

Given:

xmm6 = accumulator for y(n)
xmm5 = accumulator for y(n+1)
xmm3 = accumulator for y(n+2)
xmm2 = accumulator for y(n+3)

The four 32-bit floating point values in a register are referred to in the following discussion by a register "dot" element notation. For example:

xmm6	6.3	6.2	6.1	6.0

Algorithm:

1. Interleave the low elements of `xmm6` and `xmm3` and place them in `xmm4`.
2. Interleave the high elements of `xmm6` and `xmm3` and place them in `xmm6`.
3. Add `xmm4` and `xmm6` to obtain half of the final sums.

4. Interleave the low elements of `xmm5` and `xmm2` and place them in `xmm1`.
5. Interleave the high elements of `xmm5` and `xmm2` and place them in `xmm5`.
6. Add `xmm1` and `xmm5` to obtain the other half of the final sums.

This sequence is shown in the following diagram.

xmm4	3.1	6.1	3.0	6.0	xmm1	2.1	5.1	2.0	5.0
	+	+	+	+		+	+	+	+
xmm6	3.3	6.3	3.2	6.2	xmm5	2.3	5.3	2.2	5.2
	=	=	=	=		=	=	=	=
xmm6	3.1+3.3	6.1+6.3	3.0+3.2	6.0+6.2	xmm5	2.1+2.3	5.1+5.3	2.0+2.2	5.0+5.2

7. Interleave the low elements of `xmm6` and `xmm5` and place them in `xmm7`.
8. Interleave the high elements of `xmm6` and `xmm5` and place them in `xmm6`.
9. Add `xmm6` and `xmm7` to obtain the final output values.

These last three steps are shown below.

xmm6	2.0+2.2	3.0+3.2	5.0+5.2	6.0+6.2
	+	+	+	+
xmm7	2.1+2.3	3.1+3.3	5.1+5.3	6.1+6.3
	=	=	=	=
xmm7	y(n+3)	y(n+2)	y(n+1)	y(n)

This approach uses the full bandwidth of the `addps` instruction with each invocation.

2.2.2 A 16-Tap FIR Filter for Complex Numbers

This section discusses the design considerations that are appropriate for implementing the FIR filter for complex numbers using the Streaming SIMD Extensions.

2.2.2.1 Design Decisions

With the exception of some important, basic differences, the design of the complex version of the 16-tap FIR filter is very similar to the design discussed in Section 2.2.1 for the real 16-tap FIR filter. As is common in applications written in high-level languages that use complex numbers, a structure, consisting of a real and complex floating point value, is used to represent each complex number, and an array of structures is used to represent the input, taps, and output vectors for the filter.

Only two complex numbers can be stored in an xmm register at a time, because each complex number is represented by two floating point values, as shown below (if both the real and imaginary components of the number are stored in the register).

xmm7	Imag(1)	Real(1)	Imag(0)	Real(0)

This implies the following:

- If the same basic algorithm developed in Section 2.2.1 is used again, only two output values are generated with each loop iteration.
- Only two taps can be accessed at a time.
- Each set of two taps sits on a 16-byte boundary (if the entire taps array is 16-byte aligned).

Given the implications stated above, the taps array is changed as follows:

16 Bytes		16 Bytes		16 Bytes		16 Bytes		16 Bytes		16 Bytes		16 Bytes		16 Bytes		16 Bytes	
c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0	0	0
0	c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0	0

Each value in the preceding illustration of the taps array represents a complex number consisting of two contiguous floating point values. Note that the taps array is only copied and shifted once, as opposed to three times for the real number implementation.

Another important difference in the two implementations is the use of complex arithmetic instead of real arithmetic. A complex addition is simply the addition of the real components to produce a new real value and the addition of the imaginary components to produce a new imaginary value. However, a complex multiply is more involved.

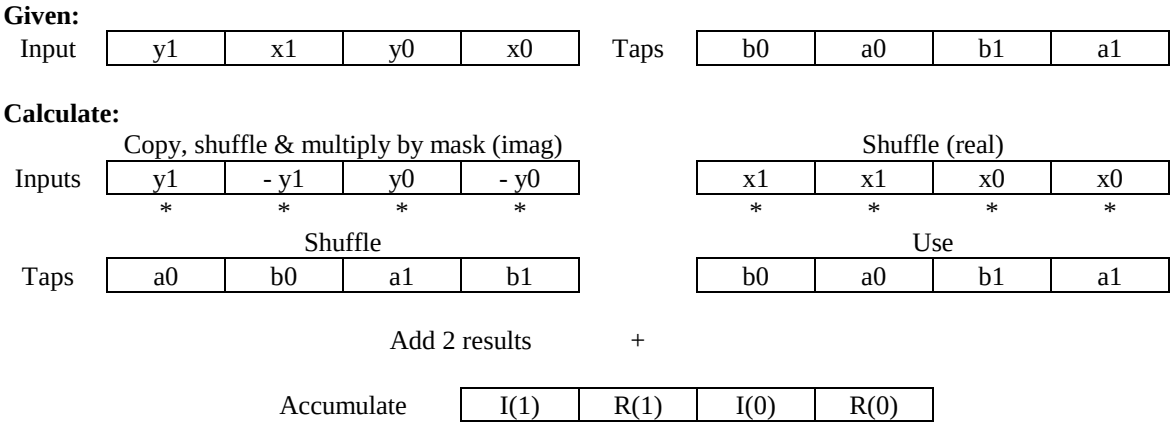
Addition

$$\begin{array}{rc} & x + yi \\ + & a + bi \\ \hline & (x+a) + (y+b)i \end{array}$$

Multiplication

$$\begin{array}{rc} & x + yi \\ * & a + bi \\ \hline & (x*a - y*b) + (x*b+y*a)i \end{array}$$

Due to the organization of the input data and the taps data, the complex multiply requires some data shuffling, two multiplies and an addition, as shown in the following diagram.



Note that two complex inputs and two complex taps are involved in the preceding calculation. The parallelism in the multiplications expressed in the real FIR filter is also present in the complex FIR filter, but the parallel multiplies calculate two complex numbers, rather than four real numbers.

At the end of each iteration, partial sums for the two complex output values are stored in two registers (accumulators). To sum "across" these registers, the contents of the registers are shuffled and added, producing two complex numbers in a single register, which can be written to the output vector in one 128-bit write.

2.3 Optimization Techniques

The following sections discuss various techniques used to optimize the performance of the code using Streaming SIMD Extensions.

2.3.1 Techniques Used for the Real Number Implementation

An unoptimized version of the real 16-tap FIR filter algorithm discussed in Section 2.2.1 was implemented using Streaming SIMD Extensions and can be found in Section 6.1. Note that the rearranging and copying of the taps array is accomplished by an outside calling routine before this code is called.

An optimized version of the real 16-tap FIR filter algorithm can be found in Section 6.2. The optimization of the Streaming SIMD Extensions implementation of the 16-tap FIR filter algorithm used many techniques which are generally applicable to optimizing code using Streaming SIMD Extensions on the P6 architecture. The following sections present several considerations that affect the optimization of the code.

2.3.1.1 Streaming SIMD Extensions

All Streaming SIMD Extensions translate to at least two micro-ops. This can make the decoder a bottleneck if lots of Streaming SIMD Extensions are being used consecutively and the resulting micro-ops retire quickly. This was not an issue for the FIR filter implementation due to the large number of memory accesses when performing the parallel multiplications and the CPU-bound nature of the interleaved additions.

2.3.1.2 The ROB and the Reservation Station

Keeping track of the number of micro-ops in the reorder buffer (ROB) and the Reservation Station proved useful in optimizing the code using Streaming SIMD Extensions. Ideally neither the ROB nor the Reservation Station should become saturated with micro-ops (limit is 40 for the ROB, 20 for the Reservation Station). Saturation of the Reservation Station indicates that micro-ops are queued up waiting for CPU resources. Usually, this situation can be eliminated through careful scheduling of instructions targeted to different CPU ports, and by taking into account instruction latencies when scheduling. Saturation of the ROB, when resulting from a Reservation Station saturation, prevents new micro-ops from being brought into the ROB, reducing the effectiveness of the out-of-order execution nature of the architecture.

2.3.1.3 Advancing Memory Loads

Memory accesses require a minimum of three clock cycles to complete if there is a cache hit on the L1 cache. These potentially long latencies should be addressed by scheduling memory accesses as early and as far away as possible from the use of the accessed data. It is also helpful to retain data accessed from

memory within the CPU for as long as possible to reduce the need to re-read the data from memory. This can be seen in the FIR filter implementation in the use of `xmm1` as a storage area to hold four input values while they are multiplied by four different sets of coefficients.

2.3.1.4 Separating Memory Accesses from Operations

Separating memory accesses from operations that use the accessed data allows the micro-ops generated to access memory to retire before the micro-ops which actually perform the operation. If a memory access is combined with an operation, all the micro-ops generated by the instruction wait to retire until the last micro-op is finished. This can leave micro-ops used to access memory waiting to retire in the ROB for multiple clocks, taking up valuable buffer space. An example of this can be seen by comparing the unoptimized code to the optimized code for performing multiplications against the coefficient data.

Unoptimized code:

```
movaps xmm0, xmm1;              // Reload [n-13:n-16] for new product
mulps xmm0, [eax + 160];        // xmm0 = input [n-13:n-16] * c2_4
```

Optimized code:

```
movaps xmm4, [eax + 160 - 32];  // Load c2_2 for new product
mulps xmm4, xmm1;               // xmm4 = input [n-5:n-8] * c2_2
```

2.3.1.5 Unrolling the Loop

The C implementation of the FIR filter has two loops: an outer loop to move upward through the input values, and an inner loop to perform the dot product between the input and taps arrays for each output value. In the implementation using Streaming SIMD Extensions, the inner loop was unrolled and only a single loop controls the function. Loop unrolling benefits performance in two ways. First, it lessens the incidence of branch misprediction by removing a conditional jump. Second, it increases the "pool" of instructions available for re-ordering and scheduling of the processor. Keep in mind that loop unrolling makes the code larger, contributing to the "code bloat" that is such a hot topic today. Performance often must be evaluated against other considerations, such as code size.

2.3.1.6 Reducing Data Dependency & Register Pressure

Comparing the unoptimized and optimized versions of the implementation using Streaming SIMD Extensions reveals several points at which registers were reallocated to reduce register pressure and increase opportunities for rescheduling instructions. The primary example of this is the use of `xmm0` to perform parallel multiplications. In the unoptimized version, `xmm0` is used exclusively to access data from the input array and perform the multiplication against the coefficient array. In the optimized version, `xmm4` and `xmm7` are brought into play to alleviate pressure from `xmm0`. While `xmm4` is used to compute values for both y(n+1) and y(n+3), the only other connection between the parallel multiplies is the use of `xmm1` to hold a copy of the input values used by the other registers. This allows for a few very precise dependencies on the parallel portion of the algorithm, and increases the opportunities for rescheduling instructions.

2.3.1.7 Wrapping the Loop Around (Software Pipelining)

The interleaved additions at the end of the loop are completely CPU-bound and very dependent upon one another. The result of this is that the ROB and the Reservation Station quickly saturate, preventing new micro-ops from entering the ROB. Due to data dependencies, the instructions could not be rescheduled very far back into the main loop body. To alleviate this condition, the first set of multiplies (against the first column of coefficients) and the loop control instructions were pulled out of the top of the loop and a copy placed at the bottom. While this increased the size of the code, the resulting opportunities for instruction scheduling prevented the saturation of the ROB and Reservation Station while improving the overall throughput of the loop. A second copy of the instructions must be placed outside the top of the loop to "prime" the loop for its first iteration.

2.3.1.8 Minimizing Pointer Arithmetic/Eliminating Unnecessary Micro-ops

For readability, much of the pointer arithmetic in the unoptimized version of the code was explicit, allowing for a detailed explanation of the accesses into the taps arrays. In the optimized version, the conversion of the explicit arithmetic to implicit address calculations contained in memory accesses reduced the number of non-essential micro-ops generated by the core of the loop. Because all of the memory accesses are by Streaming SIMD Extensions, and therefore multiple micro-op instructions, the inclusion of the address calculations did not affect the number of micro-ops the instructions generated. A generally applicable rule of thumb is to take the opportunity to eliminate unnecessary micro-ops whenever possible.

2.3.1.9 Prefetch Hints

Although input data used by the FIR filter is quite likely to be in cache, due to the fact that the data was recently accessed to build the input vector, a prefetch hint was included to pre-load the next cache line worth of data from the input array. Accesses to the taps arrays and to the historical input data occur every iteration of the loop, and therefore should maintain good temporal locality after their initial access. Keep in mind that the processor is not guaranteed to follow hints and so the performance benefits of the prefetch hint is not guaranteed.

2.3.1.10 Minimizing Cache Pollution on Write

There are issues centered around how the output vector is used, which can influence the method used to store the data. Basically, either the output vector is used (in the calling program) soon after it is populated, or it won't be accessed for some time. In the first case, the `movaps` instruction should be used to write out the data. It also may be beneficial to ensure the output vector is in the cache by using prefetch instructions, but this should be unnecessary due to the advanced write capabilities of the P6 architecture. In the second case, if the output vector won't be used for some time, it may be wise to minimize cache pollution by using the `movntps` instruction.

2.3.2 Techniques Used for the Complex Number Implementation

All of the techniques outlined in Section 2.3.1 apply to the complex 16-tap FIR filter as well. The following sections discuss a few particular areas of interest as they relate to the complex number implementation.

2.3.2.1 Unrolling the Loop

The change to the taps array increases the number of iterations of the inner loop of the basic FIR algorithm. This, combined with an increased number of instructions due to the complex multiply, results in many more instructions when the loop is unrolled. The issue of "code bloat" again must be evaluated, but is complicated by an increased number of instructions. One important note, if the loop is not unrolled, the algorithm produces a branch misprediction and pipeline stall for every iteration of the outer loop. The architecture only supports four bits of branch history (a four branch history) in its branch prediction mechanism. To reduce the size of the code, the inner loop may be unrolled only enough times to reduce the number of iterations to four. This allows the processor to "learn" the branching behavior of the code and reduce branch mispredictions while minimizing "code bloat."

2.3.2.2 Reducing Non-Value Added Instructions

It is often desirable to limit the use of shuffle, unpack, and move instructions in an algorithm, because these instructions are basically "non-value added." That is, these instructions do not perform any arithmetic function on the data. In Section 2.3.3, an alternative data storage format, geared towards parallel (or SIMD) processing is considered which eliminates the need to shuffle the complex numbers to enable complex multiplies. In the ideal case, an application should be written from the ground up using such SIMD structures. However, it is often the case that SIMD structures do not fit well with the tenets of object-orientation and the bulk of an application may not use ideal SIMD structures. Therefore, an important tradeoff in considering eliminating "non-value added" instructions is an evaluation of how much of a speedup will result from this elimination versus how much overhead is necessary to "SIMDify" the data before executing the function.

2.3.3 An Alternative Implementation of the Complex FIR Filter using a SIMD Data Structure

The definition of SIMD techniques is that a single instruction operates upon multiple data elements **of the same type**. By interleaving the real and imaginary components of complex numbers, one can argue that an instruction is not working on the same "type" of information (even though both real and complex components are represented by 32-bit floating point numbers). A more efficient version of the complex multiply can be implemented if the real and imaginary components of the complex numbers are stored separately, in their own arrays.

Given:

Complex numbers stored as structures of arrays rather than arrays of structures:

Array of Structures:

```
struct _cplx {
   float real;
   float imag;
};
struct _cplx Array[20];
```

Structure of Arrays:

```
struct _cplx {
   float *real;
   float *imag;
};
float real[20], imag[20];
struct _cplx Array;
Array.real = real;
Array.imag = imag;
```

Using a structure of arrays data format, four complex numbers can be loaded into two xmm registers. The real components are in one register while the imaginary components are in a second register. Two complex numbers then can be loaded into xmm registers and a multiplication performed, as shown in the following diagram.

Ra	R3	R2	R1	R0	Rb	R7	R6	R5	R4
Ia	I3	I2	I1	I0	Ib	I7	I6	I5	I4

Result (Rc + Ici) = [(Ra * Rb) - (Ia * Ib)] + [(Ra * Ib) + (Rb * Ia)]i

Rc	R3 * R7 - I3 * I7	R2 * R6 - I2 * I6	R1 * R5 - I1 * I5	R0 * R4 - I0 * I4
Ic	R3 * I7 + I3 * R7	R2 * I6 + I2 * R6	R1 * I5 + I1 * R5	R0 * I4 + I0 * R4

With no data shuffling, a complex multiply of four complex numbers is performed using four multiply instructions, an addition, and a subtraction, producing the resulting four complex numbers.

If this were the sole consideration, the structure of arrays implementation definitely would be preferable. However, in the case of the complex 16-tap FIR filter, a complex multiply itself is only part of the convolution. The accumulation of multiplicative results requires registers to store intermediate results. The resulting register pressure forces the structure of arrays implementation to perform more reads from memory. Because of this, the structure of arrays implementation provides only a 15% performance improvement over the array of structures implementation in Section 6.3.

A 15% performance increase is great if the structure of arrays data type is proliferated throughout an application. However, if the bulk of an application uses array of structures data types, then a conversion

from one type to the other would have to be performed before the SIMD function is called. With only a 15% performance increase, the overhead of performing this data conversion must be carefully evaluated before the structure of arrays function is used.

As a final note, the nature of SIMD functions means that a structure of arrays data type is always more efficient than an array of structures due to the removal of "data swizzling" instructions. Important considerations of how the data is used in the remainder of the application and the overhead in any data conversions that must be performed must be evaluated, however, when integrating SIMD functions into existing applications.

3. Performance

The Streaming SIMD Extensions implementation of the FIR filter was compared with both the simple C implementation discussed in this paper and an optimized x87 implementation of an FIR filter found in Intel's Signal Processing Library (SPL):

http://developer.intel.com/vtune/perflibst/spl/index.htm

As with the real 16-tap FIR filter, the implementation of the complex 16-tap FIR filter using Streaming SIMD Extensions was compared with both a simple C implementation and an optimized x87 implementation of an FIR filter found in Intel's Signal Processing Library (SPL).

3.1 Gains/Improvements

The performance gains from the implementations using Streaming SIMD Extensions technology result primarily from the SIMD processing capabilities. The gains are limited by the nature of the algorithm, which, like all algorithms which have "horizontal" dependencies across the "vertical" parallelism (e.g. dot product), require a certain number of non-value added instructions to perform tasks like shuffling and unpacking in order to implement the non-parallel portions of the algorithm.

The quality (in terms of accuracy) of the data produced by the versions of the FIR filters using Streaming SIMD Extensions, when compared with an MMX™ technology implementation using 16-bit fixed-point integers, also is greater.

3.2 Considerations

It is important to note that the implementations presented in this paper provide examples of how to implement FIR filters using Streaming SIMD Extensions. They are in no way complete "library quality" versions of these functions. Of particular importance, they do not handle border cases where the length of the input vector is not a multiple of 4 nor do they allow for an arbitrary number of taps.

4. Conclusion

This paper has presented several SIMD designs for a 16-tap FIR filter using both real and complex numbers and examined in detail the reasons for specific design decisions. The implementation using Streaming SIMD Extensions provides better accuracy than the same algorithm implemented using MMX technology instructions with 16-bit fixed-point numbers.

5. C++ Coding Example

The following is a simple implementation of an FIR filter in C++. It does not assume any history from previous executions; that is indexes into the input array which are less than zero do not contribute to the output. The only change to this algorithm for dealing with complex numbers is the transformation of the simple floating point multiply and add to their complex equivalents.

```
/* C++ implementation of FIR filter.
 *
 * The relationship between the input x(n) and output y(n) vectors of a
discrete FIR
 * filter can be described by the following equation:
 *
 *   y(n) = sum (c(k) * x(n-k)) where k varies from 0 to M - 1
 *
 * - N is the length of the input & output vectors (0 < n < N)
 * - M is the length of the filter (typically much less than N)
 * - c is a vector of coefficients or "taps" that represent the
characteristics of the
 * filter
 */
```

```
void FIR_C (float *input, int length, float *taps, int num_taps, float
*output) {
    float accum = 0.0;
  int index = 0;

  for (int n = 0; n < length; n++) {
    accum = 0.0;
    for (int m = 0; m < num_taps; m++) {
      index = n - m;
      if (index >= 0)
        accum += taps[m] * input[index];
    }
    output[n] = accum;
  }
}
```

6. Streaming SIMD Extensions Assembly Code Example

6.1 Real FIR Implementation with Streaming SIMD Extensions

```
/* This function represents an unoptimized (readable) version of the
 * Tap16_FIR_SIMD function.
 *
 * SIMD implementation of a 16-tap FIR filter.
 *
 * The relationship between the input x(n) and output y(n) vectors of a
 * discrete FIR filter can be described by the following equation:
 *
 *    y(n) = sum (c(k) * x(n-k)) where k varies from 0 to M - 1
 *
 * N is the length of the input & output vectors (0 < n < N)
 * M is the length of the filter (typically much less than N)
 * c is a vector of coefficients or "taps" that represent the
 *   characteristics of the filter
 *
 * The Tap16_FIR_SIMD function performs the convolution of two vectors at
 * the heart of a 16-tap FIR filter.  State is not explicitly maintained,
 * but can be maintained by the calling program by passing historical
 * values in the input vector, as described in 'Assumptions' below.
 *
 * Assumptions
 * -----------
 *  1. Input vector is in cache.
 *  2. The input vector is preceded by 16 (4 sets of 4) values (in order to
 *     provide a history before x(0) ).  These values should be 0.0 when
 *     running an initial input vector through the filter.  They may be used
 *     to provide state to the filter on subsequent runs by providing the
 *     last 16 values from the input vector used on the previous call to
 *     Tap16_FIR_SIMD.
 *  3. The base of all arrays (input, output, and filter data) is 16 byte
 *     aligned.
 *  4. The length of the input array MUST be a multiple of 4.
 *  5. The filter data (taps) are stored in reverse order to facilitate
 *     SIMD multiplication.
```

```
*  6. The filter data is an array of four 19-word vectors.  (Taps array)
*           Each array is a copy of the filter data which is padded with
*           zeros and aligned differently. The purpose of this is to avoid
*           misaligned accesses to the filter data. The data format is:
*
                cx_4     |     cx_3     |     cx_2     |     cx_1     |     cx_0
*                        |              |              |              |
*  c0:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[0]  |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 | 0 | 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*  c1:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[32] | 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 | 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*  c2:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[64] | 0 | 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*  c3:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[96] | 0 | 0 | 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*
* Register use
* ------------
*                 input[n]     input[n+1]      input[n+2]      input[n+3]
*                 --------     ----------      ----------      ----------
* Dot Product       xmm0          xmm0            xmm0            xmm0
* Holding           xmm1          xmm1            xmm1            xmm1
* Accumulation      xmm6          xmm5            xmm3            xmm2
* Scratch                                 xmm4
* Final Sums                              xmm7
*
*
* Notational Conventions
* ----------------------
* In the code below, several notational conventions are used to represent
* Streaming SIMD Extensions registers and their contents.
*
*    [x+3:x] - Typically used when filling an xmm
*              register from memory.  The four floats stored in
*              an xmm register are, in order from low 32 bits to
*              high, x, x+1, x+2, & x+3.
*
```

```
 *    w,z,y,x - Typically used when describing the contents of
 *              an xmm register.  The four floats stored in
 *              an xmm register are, in order from low
 *              32 bits to high, x,y,z, & w.
 *
 *    r.i     - Typically used when referencing one of the floats within
 *              an xmm register.  'r' represents the
 *              register number (e.g. for XMM1 r=1).  'i' represents the
 *              index of the float referenced in 'r' (e.g. i=0 is the float
 *              in bits 0 through 31 of register 'r', i=3 is the float in
 *              the highest 32 bits).
 *
 */

#include <assert.h>

void Tap16_FIR_SIMD_readable (float *input, int input_length,
                              float *taps, float *output)
{
//  Test assumption that all arrays are 16 byte aligned
  assert ( ((int)input & 0x0000000F) == 0 );
  assert ( ((int)taps & 0x0000000F) == 0 );
  assert ( ((int)output & 0x0000000F) == 0 );

//  Test assumption on input length = multiple of 4
  assert ( (input_length % 4) == 0 );

  __asm {
    mov esi, input;          // Set up pointers for loop
    mov edx, taps;
    mov ecx, input_length;
    mov edi, output;

    sal ecx, 2;              // ecx = length * 4 (# of bytes, float = 4)
    add esi, ecx;            // esi points to end of input array
    add esi, 64;             // Remember, the first 16 elements of the input
                             // vector are 0.0 so it's actual length is an
                             // additional 16 (16*4 = 64 bytes) elements.
    add edi, ecx;            // edi points to end of output array
    neg ecx;                 // ecx will be used as offset from beginning
                             // of arrays

// For n = {0,4,8,12,16,20,24,28,32, ...}:
```

```
  loop:
    prefetcht0 [esi + ecx + 16];   // Give the processor a hint to fetch the
                                   // next cache line of data from
                                   // the input vector.
// Initialize registers for accumulation
    xorps xmm6, xmm6;              // xmm6 = out[n]
    xorps xmm5, xmm5;              // xmm5 = out[n+1]
    xorps xmm3, xmm3;              // xmm3 = out[n+2]
    xorps xmm2, xmm2;              // xmm2 = out[n+3]

    mov eax, edx;                  // eax points to Taps array
    add eax, 64;                   // eax points to cx_0

// Calculate for input[n+3:n]
    movaps xmm0, [esi + ecx];      // Load input[n+3:n]
    movaps xmm1, xmm0;             // Save a copy of [n+3:n]
// out[n]
    mulps xmm0, [eax + 80];        // xmm0 = input [n+3:n] * c1_0
    addps xmm6, xmm0;              // accumulate
// out[n+1]
    movaps xmm0, xmm1;             // Reload [n+3:n] for new product
    mulps xmm0, [eax + 160];       // xmm0 = input [n+3:n] * c2_0
    addps xmm5, xmm0;              // accumulate
// out[n+2]
    movaps xmm0, xmm1;             // Reload [n+3:n] for new product
    mulps xmm0, [eax + 240];       // xmm0 = input [n+3:n] * c3_0
    addps xmm3, xmm0;              // accumulate

    sub eax, 16;                   // Point eax to cx_1

// out[n+3]
    movaps xmm0, xmm1;             // Reload [n+3:n] for new product
    mulps xmm0, [eax];             // xmm0 = input [n+3:n] * c0_1
    addps xmm2, xmm0;              // accumulate

// Calculate for input[n-1:n-4]
    movaps xmm0, [esi + ecx - 16]; // Load input[n-1:n-4]
    movaps xmm1, xmm0;             // Save a copy of [n-1:n-4]
// out[n]
    mulps xmm0, [eax + 80];        // xmm0 = input [n-1:n-4] * c1_1
    addps xmm6, xmm0;              // accumulate
// out[n+1]
    movaps xmm0, xmm1;             // Reload [n-1:n-4] for new product
```

```
    mulps xmm0, [eax + 160];       // xmm0 = input [n-1:n-4] * c2_1
    addps xmm5, xmm0;              // accumulate
// out[n+2]
    movaps xmm0, xmm1;             // Reload [n-1:n-4] for new product
    mulps xmm0, [eax + 240];       // xmm0 = input [n-1:n-4] * c3_1
    addps xmm3, xmm0;              // accumulate

    sub eax, 16;                   // Point eax to cx_2

// out[n+3]
    movaps xmm0, xmm1;             // Reload [n-1:n-4] for new product
    mulps xmm0, [eax];             // xmm0 = input [n-1:n-4] * c0_2
    addps xmm2, xmm0;              // accumulate

// Calculate for input[n-5:n-8]
    movaps xmm0, [esi + ecx - 32]; // Load input[n-5:n-8]
    movaps xmm1, xmm0;             // Save a copy of [n-5:n-8]
// out[n]
    mulps xmm0, [eax + 80];        // xmm0 = input [n-5:n-8] * c1_2
    addps xmm6, xmm0;              // accumulate
// out[n+1]
    movaps xmm0, xmm1;             // Reload [n-5:n-8] for new product
    mulps xmm0, [eax + 160];       // xmm0 = input [n-5:n-8] * c2_2
    addps xmm5, xmm0;              // accumulate
// out[n+2]
    movaps xmm0, xmm1;             // Reload [n-5:n-8] for new product
    mulps xmm0, [eax + 240];       // xmm0 = input [n-5:n-8] * c3_2
    addps xmm3, xmm0;              // accumulate

    sub eax, 16;                   // Point eax to cx_2

// out[n+3]
    movaps xmm0, xmm1;             // Reload [n-5:n-8] for new product
    mulps xmm0, [eax];             // xmm0 = input [n-5:n-8] * c0_3
    addps xmm2, xmm0;              // accumulate

// Calculate for input[n-9:n-12]
    movaps xmm0, [esi + ecx - 48]; // Load input[n-9:n-12]
    movaps xmm1, xmm0;             // Save a copy of [n-9:n-12]
// out[n]
    mulps xmm0, [eax + 80];        // xmm0 = input [n-9:n-12] * c1_3
    addps xmm6, xmm0;              // accumulate
// out[n+1]
```

```
    movaps xmm0, xmm1;              // Reload [n-9:n-12] for new product
    mulps xmm0, [eax + 160];        // xmm0 = input [n-9:n-12] * c2_3
    addps xmm5, xmm0;               // accumulate
// out[n+2]
    movaps xmm0, xmm1;              // Reload [n-9:n-12] for new product
    mulps xmm0, [eax + 240];        // xmm0 = input [n-9:n-12] * c3_3
    addps xmm3, xmm0;               // accumulate

    sub eax, 16;                    // Point eax to cx_2

// out[n+3]
    movaps xmm0, xmm1;              // Reload [n-9:n-12] for new product
    mulps xmm0, [eax];              // xmm0 = input [n-9:n-12] * c0_4
    addps xmm2, xmm0;               // accumulate

// Calculate for input[n-13:n-16]
    movaps xmm0, [esi + ecx - 64]; // Load input[n-13:n-16]
    movaps xmm1, xmm0;              // Save a copy of [n-13:n-16]
// out[n]
    mulps xmm0, [eax + 80];         // xmm0 = input [n-13:n-16] * c1_4
    addps xmm6, xmm0;               // accumulate
// out[n+1]
    movaps xmm0, xmm1;              // Reload [n-13:n-16] for new product
    mulps xmm0, [eax + 160];        // xmm0 = input [n-13:n-16] * c2_4
    addps xmm5, xmm0;               // accumulate
// out[n+2]
    movaps xmm0, xmm1;              // Reload [n-13:n-16] for new product
    mulps xmm0, [eax + 240];        // xmm0 = input [n-13:n-16] * c3_4
    addps xmm3, xmm0;               // accumulate

// Calculate final sums for out[n] to out[n+3]
    movaps xmm4, xmm6;              //  Copy xmm6 into xmm4
    unpcklps xmm4, xmm3;            //  xmm4 = 3.1,6.1,3.0,6.0
    unpckhps xmm6, xmm3;            //  xmm6 = 3.3,6.3,3.2,6.2
    addps xmm6, xmm4;              //  xmm6 = 3.1+3.3,6.1+6.3,3.0+3.2,6.0+6.2

    movaps xmm1, xmm5;              //  Copy xmm5 into xmm1
    unpcklps xmm1, xmm2;            //  xmm4 = 2.1,5.1,2.0,5.0
    unpckhps xmm5, xmm2;         //  xmm5 = 2.3,5.3,2.2,5.2
    addps xmm5, xmm1;            //  xmm5 = 2.1+2.3,5.1+5.3,2.0+2.2,5.0+5.2

    movaps xmm7, xmm6;           //  Copy xmm6 into xmm7
    unpcklps xmm7, xmm5;         //  xmm7 = 2.0+2.2,3.0+3.2,5.0+5.2,6.0+6.2
```

```
    unpckhps xmm6, xmm5;        //  xmm6 = 2.1+2.3,3.1+3.3,5.1+5.3,6.1+6.3
    addps xmm7, xmm6;           //  xmm7 = out[n+3],out[n+2],out[n+1],out[n]

// Finished calculating out[n+3:n]
//
//   There are some issues centered around how the output vector will be
//   used which can influence the method used to store the data.  Basically,
//   either the output vector will be used (in the calling program) soon
//   after it is populated or it won't be accessed for some time.
//
//   In the first case, the MOVAPS instruction should be used.  It may be
//   beneficial to ensure the output vector is in the cache using prefetch
//   instructions.
//
//   In the second case, if the output vector won't be used for some time,
//   it may be wise to minimize cache pollution by using the MOVNTPS
//   instruction.
//
    movaps [edi+ecx], xmm7;  // Move out[n+3:n] (xmm7) to out[n+3:n]

    add ecx, 16;             // Increment n to n+4
    jnz loop;                // Exit when ecx reaches 0 (end of input array)
  }
}
```

6.2 Optimized Real FIR Implementation with Streaming SIMD Extensions

```
/* This function represents an optimized version of the Tap16_FIR_SIMD
 * function.
 * SIMD implementation of a 16-tap FIR filter.
 *
 * The relationship between the input x(n) and output y(n) vectors of a
 * discrete FIR filter can be described by the following equation:
 *
 *    y(n) = sum (c(k) * x(n-k)) where k varies from 0 to M - 1
 *
 * N is the length of the input & output vectors (0 < n < N)
 * M is the length of the filter (typically much less than N)
 * c is a vector of coefficients or "taps" that represent the
 *   characteristics of the filter
 *
 * The Tap16_FIR_SIMD function performs the convolution of two vectors at
 * the heart of a 16-tap FIR filter.  State is not explicitly maintained,
 * but can be maintained by the calling program by passing historical
 * values in the input vector, as described in 'Assumptions' below.
 *
 * Assumptions
 * -----------
 *  1. Input vector is in cache.
 *  2. The input vector is preceded by 16 (4 sets of 4) values (in order to
 *     provide a history before x(0) ).  These values should be 0.0 when
 *     running an initial input vector through the filter.  They may be used
 *     to provide state to the filter on subsequent runs by providing the
 *     last 16 values from the input vector used on the previous call to
 *     Tap16_FIR_SIMD.
 *  3. The base of all arrays (input, output, and filter data) is 16 byte
 *     aligned.
 *  4. The length of the input array MUST be a multiple of 4.
 *  5. The filter data (taps) are stored in reverse order to facilitate
 *     SIMD multiplication.
```

```
*  6. The filter data is an array of four 19-word vectors.  (Taps array)
*           Each array is a copy of the filter data which is padded with
*           zeros and aligned differently. The purpose of this is to avoid
*           misaligned accesses to the filter data. The data format is:
*
                cx_4     |     cx_3     |     cx_2     |     cx_1     |     cx_0
*                        |              |              |              |
*  c0:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[0]  |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 | 0 | 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*  c1:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[32] | 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 | 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*  c2:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[64] | 0 | 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*  c3:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[96] | 0 | 0 | 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 |
*           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                        |              |              |              |
*
* Register use
* ------------
*                 input[n]     input[n+1]      input[n+2]      input[n+3]
*                 --------     ----------      ----------      ----------
* Dot Product       xmm0          xmm0            xmm0            xmm0
* Holding           xmm1          xmm1            xmm1            xmm1
* Accumulation      xmm6          xmm5            xmm3            xmm2
* Scratch                                 xmm4
* Final Sums                              xmm7
*
*
* Notational Conventions
* ----------------------
* In the code below, several notational conventions are used to represent
* xmm registers and their contents.
*
*    [x+3:x] - Typically used when filling an xmm
*              register from memory.  The four floats stored in
*              an xmm register are, in order from low 32 bits to
*              high, x, x+1, x+2, & x+3.
*
```

```
 *    w,z,y,x - Typically used when describing the contents of
 *              an xmm register.  The four floats stored in
 *              an xmm register are, in order from low
 *              32 bits to high, x,y,z, & w.
 *
 *    r.i     - Typically used when referencing one of the floats within
 *              an xmm register.  'r' represents the
 *              register number (e.g. for XMM1 r=1).  'i' represents the
 *              index of the float referenced in 'r' (e.g. i=0 is the float
 *              in bits 0 through 31 of register 'r', i=3 is the float in
 *              the highest 32 bits).
 *
 */

#include <assert.h>

void Tap16_FIR_SIMD (float *input, int input_length,
                     float *taps, float *output)
{

// Test assumption that all arrays are 16 byte aligned
  assert ( ((int)input & 0x0000000F) == 0 );
  assert ( ((int)taps & 0x0000000F) == 0 );
  assert ( ((int)output & 0x0000000F) == 0 );

// Test assumption on input length = multiple of 4
  assert ( (input_length % 4) == 0 );

  __asm {
    mov ecx, input_length;   // Set up pointers for loop
    sal ecx, 2;              // ecx = length * 4 (# of bytes, float = 4)
    mov esi, input;
    add esi, ecx;            // esi points to end of input array
    mov edx, taps;
    add esi, 64;             // Remember, the first 16 elements of the input
                             // vector are 0.0 so it's actual length is an
                             // additional 16 (16*4 = 64 bytes) elements.
    mov edi, output;
    add edi, ecx;            // edi points to end of output array
    neg ecx;                 // ecx will be used as offset from beginning of
                             // arrays

//Pulled out of loop to facilitate scheduling instructions at loop boundary.
```

```
//Calculate for input[n+3:n]
    movaps xmm0, [esi + ecx];        // Load input[n+3:n]
    mov eax, edx;                    // eax points to Taps array
    add eax, 64;                     // eax points to cx_0
    movaps xmm4, [eax + 160];        // Load c2_0 for new product
    movaps xmm2, [eax + 240];        // Load c3_0 for new product
    movaps xmm1, xmm0;               // Save a copy of [n+3:n]
// out[n]
    mulps xmm0, [eax + 80];          // xmm0 = input [n+3:n] * c1_0
// out[n+1]
    mulps xmm4, xmm1;                // xmm4 = input [n+3:n] * c2_0
// out[n+2]
    mulps xmm2, xmm1;                // xmm2 = input [n+3:n] * c3_0

// For n = {0,4,8,12,16,20,24,28,32, ...}:
  loop:
// Calculate for input[n+3:n]
    xorps xmm6, xmm6;               // xmm6 = out[n]
    prefetcht0 [esi + ecx + 16];    // Give the processor a hint to fetch
                                    // the next cache line of data from the
                                    // input vector.
    xorps xmm5, xmm5;               // xmm5 = out[n+1]
    xorps xmm3, xmm3;               // xmm3 = out[n+2]
    addps xmm6, xmm0;               // accumulate
    movaps xmm0, [esi + ecx - 16];  // Load input[n-1:n-4]
    addps xmm5, xmm4;               // accumulate
    movaps xmm4, [eax - 16];        // Load c0_1 for new product
    addps xmm3, xmm2;               // accumulate
    movaps xmm7, [eax + 240 - 16];  // Load c3_1 for new product
    xorps xmm2, xmm2;               // xmm2 = out[n+3]
// out[n+3]
    mulps xmm4, xmm1;               // xmm4 = input [n+3:n] * c0_1

// Calculate for input[n-1:n-4]
    movaps xmm1, xmm0;              // Save a copy of [n-1:n-4]
// out[n]
    mulps xmm0, [eax + 80 - 16];    // xmm0 = input [n-1:n-4] * c1_1
    addps xmm2, xmm4;               // accumulate
    movaps xmm4, [eax + 160 - 16];  // Load c2_1 for new product
// out[n+2]
    mulps xmm7, xmm1;               // xmm7 = input [n-1:n-4] * c3_1
    addps xmm6, xmm0;               // accumulate
    movaps xmm0, [esi + ecx - 32];  // Load input[n-5:n-8]
```

```
// out[n+1]
    mulps xmm4, xmm1;               // xmm4 = input [n-1:n-4] * c2_1
    addps xmm5, xmm4;               // accumulate
    movaps xmm4, [eax - 32];        // Load c0_2 for new product
    addps xmm3, xmm7;               // accumulate
    movaps xmm7, [eax + 240 - 32];  // Load c3_2 for new product
// out[n+3]
    mulps xmm4, xmm1;               // xmm4 = input [n-1:n-4] * c0_2

// Calculate for input[n-5:n-8]
    movaps xmm1, xmm0;              // Save a copy of [n-5:n-8]
// out[n]
    mulps xmm0, [eax + 80 - 32];    // xmm0 = input [n-5:n-8] * c1_2
    addps xmm2, xmm4;               // accumulate
    movaps xmm4, [eax + 160 - 32];  // Load c2_2 for new product
// out[n+2]
    mulps xmm7, xmm1;               // xmm7 = input [n-5:n-8] * c3_2
    addps xmm6, xmm0;               // accumulate
    movaps xmm0, [esi + ecx - 48];  // Load input[n-9:n-12]
// out[n+1]
    mulps xmm4, xmm1;               // xmm4 = input [n-5:n-8] * c2_2
    addps xmm5, xmm4;               // accumulate
    movaps xmm4, [eax - 48];        // Load c0_3 for new product
    addps xmm3, xmm7;               // accumulate
    movaps xmm7, [eax + 240 - 48];  // Load c3_3 for new product
// out[n+3]
    mulps xmm4, xmm1;               // xmm4 = input [n-5:n-8] * c0_3

// Calculate for input[n-9:n-12]
    movaps xmm1, xmm0;              // Save a copy of [n-9:n-12]
// out[n]
    mulps xmm0, [eax + 80 - 48];    // xmm0 = input [n-9:n-12] * c1_3
    addps xmm2, xmm4;               // accumulate
    movaps xmm4, [eax + 160 - 48];  // Load c2_3 for new product
// out[n+2]
    mulps xmm7, xmm1;               // xmm7 = input [n-9:n-12] * c3_3
    addps xmm6, xmm0;               // accumulate
    movaps xmm0, [esi + ecx - 64];  // Load input[n-13:n-16]
```

```
// out[n+1]
    mulps xmm4, xmm1;               // xmm4 = input [n-9:n-12] * c2_3
    addps xmm5, xmm4;               // accumulate
    movaps xmm4, [eax - 64];        // Load c0_4 for new product
    addps xmm3, xmm7;               // accumulate
// out[n+3]
    mulps xmm4, xmm1;               // xmm4 = input [n-9:n-12] * c0_4

// Calculate for input[n-13:n-16]
    movaps xmm1, xmm0;              // Save a copy of [n-13:n-16]
// out[n]
    mulps xmm0, [eax + 80 - 64];    // xmm0 = input [n-13:n-16] * c1_4
    movaps xmm7, [eax + 240 - 64];  // Load c3_4 for new product
    addps xmm2, xmm4;               // accumulate
    movaps xmm4, [eax + 160 - 64];  // Load c2_4 for new product
// out[n+2]
    mulps xmm7, xmm1;               // xmm0 = input [n-13:n-16] * c3_4
    addps xmm6, xmm0;               // accumulate
// out[n+1]
    mulps xmm4, xmm1;               // xmm4 = input [n-13:n-16] * c2_4
    addps xmm5, xmm4;               // accumulate
    movaps xmm0, [esi + ecx + 16];  // Load input[n+3:n]
// Calculate final sums for out[n] to out[n+3]
    addps xmm3, xmm7;               // accumulate
    movaps xmm4, xmm6;              // Copy xmm6 into xmm4
    movaps xmm1, xmm5;              // Copy xmm5 into xmm1
    unpcklps xmm4, xmm3;            // xmm4 = 3.1,6.1,3.0,6.0
    unpckhps xmm5, xmm2;            // xmm5 = 2.3,5.3,2.2,5.2
    mov eax, edx;                   // eax points to Taps array
    add eax, 64;                    // eax points to cx_0
    unpcklps xmm1, xmm2;            // xmm4 = 2.1,5.1,2.0,5.0

    movaps xmm2, [eax + 240];       // Load c3_0 for new product
    unpckhps xmm6, xmm3;            // xmm6 = 3.3,6.3,3.2,6.2
    addps xmm6, xmm4;              // xmm6 = 3.1+3.3,6.1+6.3,3.0+3.2,6.0+6.2
    movaps xmm4, [eax + 160];       // Load c2_0 for new product

    addps xmm5, xmm1;              // xmm5 = 2.1+2.3,5.1+5.3,2.0+2.2,5.0+5.2
    movaps xmm7, xmm6;             // Copy xmm6 into xmm7
    movaps xmm1, xmm0;             // Save a copy of [n+3:n]

    unpcklps xmm7, xmm5;           // xmm7 = 2.0+2.2,3.0+3.2,5.0+5.2,6.0+6.2
// out[n+2]
```

```
    mulps xmm2, xmm1;              // xmm2 = input [n+3:n] * c3_0
    unpckhps xmm6, xmm5;           // xmm6 = 2.1+2.3,3.1+3.3,5.1+5.3,6.1+6.3
// out[n]
    mulps xmm0, [eax + 80];        // xmm0 = input [n+3:n] * c1_0
    addps xmm7, xmm6;            // xmm7 = out[n+3],out[n+2],out[n+1],out[n]
// out[n+1]
    mulps xmm4, xmm1;              // xmm4 = input [n+3:n] * c2_0

// Finished calculating out[n+3:n]
//
//   There are some issues centered around how the output vector will be
//   used which can influence the method used to store the data.  Basically,
//   either the output vector will be used (in the calling program) soon
//   after it is populated or it won't be accessed for some time.
//
//   In the first case, the MOVAPS instruction should be used.  It may be
//   beneficial to ensure the output vector is in the cache using prefetch
//   instructions.
//
//   In the second case, if the output vector won't be used for some time,
//   it may be wise to minimize cache pollution by using the MOVNTPS
//   instruction.
//
    movaps [edi+ecx], xmm7;  // Move out[n+3:n] (xmm7) to out[n+3:n]

    add ecx, 16;             // Increment n to n+4
    jnz loop;                // Exit when ecx reaches 0 (end of input array)
  }
}
```

6.3 Complex FIR Implementation with Streaming SIMD Extensions

```
/* This listing contains the UNoptimized version.
 *
 * Streaming SIMD Extensions implementation of FIR filter for complex
numbers
 * using 16 taps and an input/output size of 40.
 *
 * The relationship between the input x(n) and output y(n) vectors of a
 * discrete FIR filter can be described by the following equation:
 *
 *    y(n) = sum (c(k) * x(n-k)) where k varies from 0 to M - 1
 *
```

```
* N is the length of the input & output vectors (0 < n < N)
* M is the length of the filter (typically much less than N)
* c is a vector of coefficients or "taps" that represent the
*   characteristics of the filter
*
* Remember:
*   Because this algorithm works on complex numbers, every variable is made
*   up of two floats (representing the real and imaginary components of the
*   number) and takes up 8 bytes of space (4 bytes per float).
*
* Assumptions:
*
*  1. The input vector is preceded by 16 (8 sets of 2) complex values (in
*     order to provide a history before x(0)).  These values should be
*     0.0+0.0i when initializing the filter.
*  2. The base of all arrays (input, output, and filter data) is 16 byte
*     aligned.
*  3. The length of the input array MUST be a multiple of 2.
*  4. The filter data (taps) are stored in reverse order to facilitate SIMD
*     multiplication.
*  5. The filter data is an array of four 19-element vectors.
*           Each array is a copy of the filter data which is padded with
*           zeros and aligned differently. The purpose of this is to avoid
*           misaligned accesses to the filter data. The data format is:
*
*
*            cx_8    cx_7    cx_6    cx_5    cx_4    cx_3    cx_2    cx_1    cx_0
*                 |       |       |       |       |       |       |       |
*  c0:     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[0] |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 |
*          +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*                |       |       |       |       |       |       |       |
*  c1:     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*  src2[32]| 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 |
*          +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*
*  Register use:
*                       out[n]                          out[n+1]
*              input[n]     input[n+1]        input[n]      input[n+1]
*              --------     ----------        --------      ----------
* Load         <---------xmm0--------->     <----------xmm1---------->
* Scratch      <---------xmm2--------->     <----------xmm3---------->
* Result       <---------xmm4--------->     <----------xmm5---------->
* Final Sums   <---------xmm6--------->     <----------xmm7---------->
```

```
 */

#include <assert.h>

const float neg_mask[4] = {-1.0, 1.0, -1.0, 1.0};

void Tap16_FIRc_SIMD_orig (Complex *input, int input_length,
                           Complex *taps, Complex *output)
{
// Test assumption that all arrays are 16 byte aligned
  assert ( ((int)input & 0x0000000F) == 0 );
  assert ( ((int)taps & 0x0000000F) == 0 );
  assert ( ((int)output & 0x0000000F) == 0 );

// Test assumption on input length = multiple of 2
  assert ( (input_length % 2) == 0 );

  __asm {
    mov esi, input;          // Set up pointers for loop
    mov edx, taps;
    mov ecx, input_length;
    mov edi, output;

    sal ecx, 3;              // ecx = length * 8 (# of bytes, complex = 8)
    add esi, ecx;            // esi points to end of input array
    add esi, 128;            // Remember, the first 16 elements of the input
                             // vector are 0.0 so it's actual length is an
                             // additional 128 (16*8 = 128 bytes) elements.
    add edi, ecx;            // edi points to end of output array
    neg ecx;                 // ecx will be used as offset from beginning
                             // of arrays
// For n = {0,2,4,8, ...}:
  loop:
    prefetcht0 [esi+ecx+16];  // Prefetch next inputs into cache

    mov eax, 128;            // edx + eax points to cx_0
    mov ebx, ecx;            // esi + ebx points to current input values
```

```
// Initialize registers for accumulation
    xorps xmm6, xmm6;        // xmm6 will hold sums for out[n]
    xorps xmm7, xmm7;        // xmm7 will hold sums for out[n+1]

inner_loop:
// Calculate out[n]
    movaps xmm0, [esi + ebx];        // Load input[n+1:n]: y1,x1,y0,x1
    movaps xmm2, [edx + eax + 144];  // Load c1_0 from taps: b1,a1,b0,a0

    movaps xmm1, xmm0;               // Copy input[n+1:n] into xmm1
    shufps xmm1, xmm1, 0xA0;         // Shuffle inputs: x1,x1,x0,x0 (real)

    shufps xmm0, xmm0, 0xF5;         // Shuffle inputs: y1,y1,y0,y0 (imag)
    mulps xmm0, neg_mask;            // xmm0 = y1,-y1,y0,-y0

    movaps xmm4, xmm2;               // Copy taps into xmm4
    shufps xmm4, xmm4, 0xB1;         // xmm4 = a1,b1,a0,b0

    mulps xmm2, xmm1;                 // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;                 // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1

    addps xmm4, xmm2;                 // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                      //        x0*b1+y0*a1,x0*a1-y0*b1
    addps xmm6, xmm4;                 // Accumulate into xmm6

// Calculate out[n+1]
    movaps xmm3, [edx + eax - 16];    // Load c0_x from taps: b1,a1,b0,a0

    movaps xmm5, xmm3;                // Copy taps into xmm5
    shufps xmm5, xmm5, 0xB1;          // xmm5 = a1,b1,a0,b0

    mulps xmm3, xmm1;                 // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm5, xmm0;                 // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1

    addps xmm5, xmm3;                 // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                      //        x0*b1+y0*a1,x0*a1-y0*b1
    addps xmm7, xmm5;                 // Accumulate into xmm7

    sub ebx, 16;       // esi + ebx points to previous 2 input values
    sub eax, 16;       // edx + eax points to cx_(-1)
    jg inner_loop;     // Continue inner_loop until front of taps is reached

// Calculate last out[n] value
```

```
    movaps xmm0, [esi + ebx];         // Load previous 2 inputs: y1,x1,y0,x1
    movaps xmm2, [edx + eax + 144];   // Load c1_0 from taps: b1,a1,b0,a0

    movaps xmm1, xmm0;                // Copy inputs into xmm1
    shufps xmm1, xmm1, 0xA0;          // Shuffle inputs: x1,x1,x0,x0 (real)

    shufps xmm0, xmm0, 0xF5;          // Shuffle inputs: y1,y1,y0,y0 (imag)
    mulps xmm0, neg_mask;             // xmm0 = y1,-y1,y0,-y0

    movaps xmm4, xmm2;                // Copy taps into xmm4
    shufps xmm4, xmm4, 0xB1;          // xmm4 = a1,b1,a0,b0

    mulps xmm2, xmm1;                 // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;                 // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1

    addps xmm4, xmm2;                 // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                      //        x0*b1+y0*a1,x0*a1-y0*b1
    addps xmm6, xmm4;                 // Accumulate into xmm6

// xmm6 = a0_i1, a0_r1, a0_i0, a0_r0 (sums for out[n])
// xmm7 = a1_i1, a1_r1, a1_i0, a1_r0 (sums for out[n+1])
//
// Add across xmm6 (out[n]) & xmm7 (out[n+1]) and place results in xmm6
    movaps xmm0, xmm6;                // Copy sums for out[n] into xmm0
    shufps xmm6, xmm7, 0x44;          // xmm6 = a1_i0, a1_r0, a0_i0, a0_r0
    shufps xmm0, xmm7, 0xEE;          // xmm0 = a1_i1, a1_r1, a0_i1, a0_r1

    addps xmm6, xmm0;                 // xmm6 = out[n+1:n]

    movaps [edi + ecx], xmm6;         // Output out[n+1:n]

    add ecx, 16;                      // Increment n to n+4
    jnz loop;                         // Exit when ecx reaches 0
                                      // (end of input array)
  }
}
```

6.4 Optimized Complex FIR Implementation with Streaming SIMD Extensions

```
/* This listing contains the Optimized version.
 *
 * Streaming SIMD Extensions implementation of FIR filter for complex
numbers
 * using 16 taps and an input/output size of 40.
 *
 * The relationship between the input x(n) and output y(n) vectors of a
 * discrete FIR filter can be described by the following equation:
 *
 *    y(n) = sum (c(k) * x(n-k)) where k varies from 0 to M - 1
 *
 * N is the length of the input & output vectors (0 < n < N)
 * M is the length of the filter (typically much less than N)
 * c is a vector of coefficients or "taps" that represent the
 *   characteristics of the filter
 *
 * Remember:
 *   Because this algorithm works on complex numbers, every variable is made
 *   up of two floats (representing the real and imaginary components of the
 *   number) and takes up 8 bytes of space (4 bytes per float).
 *
 * Assumptions:
 *
 *  1. The input vector is preceded by 16 (8 sets of 2) complex values (in
 *     order to provide a history before x(0)).  These values should be
 *     0.0+0.0i when initializing the filter.
 *  2. The base of all arrays (input, output, and filter data) is 16 byte
 *     aligned.
 *  3. The length of the input array MUST be a multiple of 2.
 *  4. The filter data (taps) are stored in reverse order to facilitate SIMD
 *     multiplication.
 *  5. The filter data is an array of four 19-element vectors.
 *           Each array is a copy of the filter data which is padded with
 *           zeros and aligned differently. The purpose of this is to avoid
 *           misaligned accesses to the filter data. The data format is:
 *
 *
```

```
 *            cx_8    cx_7    cx_6    cx_5    cx_4    cx_3    cx_2    cx_1    cx_0
 *                  |       |       |       |       |       |       |       |
 *  c0:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 *  src2[0]  |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 | 0 |
 *           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 *                  |       |       |       |       |       |       |       |
 *  c1:      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 *  src2[32] | 0 |c15|c14|c13|c12|c11|c10|c09|c08|c07|c06|c05|c04|c03|c02|c01|c00| 0 |
 *           +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 *
 *   Register use:
 *                        out[n]                          out[n+1]
 *                input[n]     input[n+1]         input[n]      input[n+1]
 *                --------     ----------         --------      ----------
 * Load           <---------xmm0--------->       <----------xmm1---------->
 * Scratch        <---------xmm2--------->       <----------xmm3---------->
 * Result         <---------xmm4--------->       <----------xmm5---------->
 * Final Sums     <---------xmm6--------->       <----------xmm7---------->
 */
#include <assert.h>

const float neg_mask[4] = {-1.0, 1.0, -1.0, 1.0};

void Tap16_FIRc_SIMD (Complex *input, int input_length,
                      Complex *taps, Complex *output)
{
// Test assumption that all arrays are 16 byte aligned
  assert ( ((int)input & 0x0000000F) == 0 );
  assert ( ((int)taps & 0x0000000F) == 0 );
  assert ( ((int)output & 0x0000000F) == 0 );

// Test assumption on input length = multiple of 2
  assert ( (input_length % 2) == 0 );

  __asm {
    mov esi, input;          // Set up pointers for loop
    mov edx, taps;
    mov ecx, input_length;
    mov edi, output;

    sal ecx, 3;              // ecx = length * 8 (# of bytes, complex = 8)
    add esi, ecx;            // esi points to end of input array
```

```
    add esi, 128;            // Remember, the first 16 elements of the input
                             // vector are 0.0 so it's actual length is an
                             // additional 128 (16*8 = 128 bytes) elements.
    add edi, ecx;            // edi points to end of output array
    neg ecx;                 // ecx will be used as offset from beginning
                             // of arrays

// The following code "primes the loop" for increased
// scheduling opportunities
    movaps xmm7, [esi + ecx];     // Load input[n+1:n]: y1,x1,y0,x1
    movaps xmm2, [edx + 272];     // Load c1_0 from taps: b1,a1,b0,a0
    movaps xmm1, xmm7;            // Copy input[n+1:n] into xmm1
    movaps xmm3, [edx + 112];     // Load c0_x from taps: b1,a1,b0,a0

    shufps xmm7, xmm7, 0xF5;      // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;            // Copy taps into xmm4

    mulps xmm7, neg_mask;         // xmm7 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;      // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;            // Copy taps into xmm5

    shufps xmm4, xmm4, 0xB1;      // xmm4 = a1,b1,a0,b0

    shufps xmm5, xmm5, 0xB1;      // xmm5 = a1,b1,a0,b0

// For n = {0,2,4,8, ...}:
  loop:
    prefetcht0 [esi+ecx+16];         // Prefetch next inputs into cache
// Calculate out[n] & out[n+1] for input[n+1:n]
    mulps xmm2, xmm1;              // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm7;              // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    xorps xmm6, xmm6;              // xmm6 will hold sums for out[n]
    mulps xmm3, xmm1;              // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1

    mulps xmm5, xmm7;              // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1
    xorps xmm7, xmm7;              // xmm7 will hold sums for out[n+1]
    movaps xmm0, [esi + ecx - 16]; // Load input[n+1:n]: y1,x1,y0,x1

    addps xmm4, xmm2;              // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                   //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 256];      // Load c1_0 from taps: b1,a1,b0,a0

// Calculate out[n] & out[n+1] for input[n-1:n-2]
```

```
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy input[n+1:n] into xmm1
    addps xmm6, xmm4;               // Accumulate into xmm6

    movaps xmm3, [edx + 96];        // Load c0_x from taps: b1,a1,b0,a0

    addps xmm7, xmm5;               // Accumulate into xmm7

    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4

    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;              // Copy taps into xmm5

    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0

    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0

    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm3, xmm1;               // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1

    mulps xmm5, xmm0;               // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1
    movaps xmm0, [esi + ecx - 32];  // Load input[n+1:n]: y1,x1,y0,x1
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 240];       // Load c1_0 from taps: b1,a1,b0,a0

// Calculate out[n] & out[n+1] for input[n-3:n-4]
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy input[n+1:n] into xmm1
    addps xmm6, xmm4;               // Accumulate into xmm6

    movaps xmm3, [edx + 80];        // Load c0_x from taps: b1,a1,b0,a0

    addps xmm7, xmm5;               // Accumulate into xmm7

    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4
```

```
    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;              // Copy taps into xmm5

    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0
    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0

    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm3, xmm1;               // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm5, xmm0;               // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1

    movaps xmm0, [esi + ecx - 48];  // Load input[n+1:n]: y1,x1,y0,x1
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 224];       // Load c1_0 from taps: b1,a1,b0,a0
// Calculate out[n] & out[n+1] for input[n-5:n-6]
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy input[n+1:n] into xmm1

    addps xmm6, xmm4;               // Accumulate into xmm6

    movaps xmm3, [edx + 64];        // Load c0_x from taps: b1,a1,b0,a0

    addps xmm7, xmm5;               // Accumulate into xmm7

    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4

    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;              // Copy taps into xmm5

    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0
    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0

    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm3, xmm1;               // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm5, xmm0;               // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1
```

```
    movaps xmm0, [esi + ecx - 64];  // Load input[n+1:n]: y1,x1,y0,x1
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 208];       // Load c1_0 from taps: b1,a1,b0,a0
// Calculate out[n] & out[n+1] for input[n-7:n-8]
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy input[n+1:n] into xmm1
    addps xmm6, xmm4;               // Accumulate into xmm6

    movaps xmm3, [edx + 48];        // Load c0_x from taps: b1,a1,b0,a0

    addps xmm7, xmm5;               // Accumulate into xmm7

    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4

    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;              // Copy taps into xmm5

    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0
    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0

    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm3, xmm1;               // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm5, xmm0;               // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1

    movaps xmm0, [esi + ecx - 80];  // Load input[n+1:n]: y1,x1,y0,x1
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 192];       // Load c1_0 from taps: b1,a1,b0,a0
// Calculate out[n] & out[n+1] for input[n-9:n-10]
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy input[n+1:n] into xmm1

    addps xmm6, xmm4;               // Accumulate into xmm6

    movaps xmm3, [edx + 32];        // Load c0_x from taps: b1,a1,b0,a0

    addps xmm7, xmm5;               // Accumulate into xmm7
```

```
    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4

    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;              // Copy taps into xmm5
    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0

    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0

    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm3, xmm1;               // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm5, xmm0;               // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1

    movaps xmm0, [esi + ecx - 96];  // Load input[n+1:n]: y1,x1,y0,x1
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 176];       // Load c1_0 from taps: b1,a1,b0,a0
// Calculate out[n] & out[n+1] for input[n-11:n-12]
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy input[n+1:n] into xmm1
    addps xmm6, xmm4;               // Accumulate into xmm6
    movaps xmm3, [edx + 16];        // Load c0_x from taps: b1,a1,b0,a0
    addps xmm7, xmm5;               // Accumulate into xmm7

    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4

    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;              // Copy taps into xmm5
    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0
    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0

    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm3, xmm1;               // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm5, xmm0;               // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1

    movaps xmm0, [esi + ecx - 112]; // Load input[n+1:n]: y1,x1,y0,x1
```

```
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 160];       // Load c1_0 from taps: b1,a1,b0,a0

// Calculate out[n] & out[n+1] for input[n-13:n-14]
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy input[n+1:n] into xmm1
    addps xmm6, xmm4;               // Accumulate into xmm6

    movaps xmm3, [edx];             // Load c0_x from taps: b1,a1,b0,a0
    addps xmm7, xmm5;               // Accumulate into xmm7

    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4
    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm5, xmm3;              // Copy taps into xmm5
    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0
    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0

    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm3, xmm1;               // xmm3 = x1*b0,  x1*a0, x0*b1,  x0*a1
    mulps xmm5, xmm0;               // xmm5 = y1*a0, -y1*b0, y0*a1, -y0*b1

    movaps xmm0, [esi + ecx - 128]; // Load previous 2 inputs: y1,x1,y0,x1
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 144];       // Load c1_0 from taps: b1,a1,b0,a0

// Calculate last out[n] value (for input [n-15:n-16]
    addps xmm5, xmm3;               // xmm5 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm1, xmm0;              // Copy inputs into xmm1
    addps xmm6, xmm4;               // Accumulate into xmm6

    shufps xmm0, xmm0, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    movaps xmm4, xmm2;              // Copy taps into xmm4
    mulps xmm0, neg_mask;           // xmm0 = y1,-y1,y0,-y0
    addps xmm7, xmm5;               // Accumulate into xmm7

    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0
```

```
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)
    movaps xmm3, [edx + 112];       // Load c0_x from taps: b1,a1,b0,a0

    mulps xmm4, xmm0;               // xmm4 = y1*a0, -y1*b0, y0*a1, -y0*b1
    mulps xmm2, xmm1;               // xmm2 = x1*b0,  x1*a0, x0*b1,  x0*a1
    movaps xmm5, xmm3;              // Copy taps into xmm5
    addps xmm4, xmm2;               // xmm4 = x1*b0+y1*a0,x1*a0-y1*b0,
                                    //        x0*b1+y0*a1,x0*a1-y0*b1
    movaps xmm2, [edx + 272];       // Load c1_0 from taps: b1,a1,b0,a0
    shufps xmm5, xmm5, 0xB1;        // xmm5 = a1,b1,a0,b0
    addps xmm6, xmm4;               // Accumulate into xmm6

// xmm6 = a0_i1, a0_r1, a0_i0, a0_r0 (sums for out[n])
// xmm7 = a1_i1, a1_r1, a1_i0, a1_r0 (sums for out[n+1])
//
// Add across xmm6 (out[n]) & xmm7 (out[n+1]) and place results in xmm6
    movaps xmm0, xmm6;              // Copy sums for out[n] into xmm0
    shufps xmm6, xmm7, 0x44;        // xmm6 = a1_i0, a1_r0, a0_i0, a0_r0
    movaps xmm4, xmm2;              // Copy taps into xmm4
    shufps xmm0, xmm7, 0xEE;        // xmm0 = a1_i1, a1_r1, a0_i1, a0_r1
    movaps xmm7, [esi + ecx + 16];  // Load input[n+1:n]: y1,x1,y0,x1
    addps xmm6, xmm0;               // xmm6 = out[n+1:n]
    shufps xmm4, xmm4, 0xB1;        // xmm4 = a1,b1,a0,b0

    movaps xmm1, xmm7;              // Copy input[n+1:n] into xmm1
    shufps xmm7, xmm7, 0xF5;        // Shuffle inputs: y1,y1,y0,y0 (imag)
    shufps xmm1, xmm1, 0xA0;        // Shuffle inputs: x1,x1,x0,x0 (real)

    mulps xmm7, neg_mask;           // xmm7 = y1,-y1,y0,-y0
    movaps [edi + ecx], xmm6;       // Output out[n+1:n]

    add ecx, 16;                    // Increment n to n+4
    jnz loop;                       // Exit when ecx reaches 0
                                    // (end of input array)
  }
}
```